

# Itinéraire entre deux stations du métro parisien

*Kim Antunez et Alain Quartier-la-Tente*

*07/01/2020 - 15h30 à 15h45*



## Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Le programme</b>	<b>2</b>
2.1	Prise en main . . . . .	2
2.2	Démonstration . . . . .	2
<b>3</b>	<b>Description des données</b>	<b>6</b>
3.1	Description des données à disposition . . . . .	6
3.2	Difficultés et solutions adoptées . . . . .	6
<b>4</b>	<b>Description des classes</b>	<b>8</b>
4.1	Classes liées au réseau de métro parisien . . . . .	8
4.2	Classes liées à l'algorithme du plus court chemin . . . . .	8
4.3	Classe faisant le lien entre l'algorithme et les données . . . . .	9
4.4	Classe générant l'Interface Homme-Machine . . . . .	10
<b>5</b>	<b>L'algorithme de Dijkstra</b>	<b>10</b>
<b>6</b>	<b>Pour conclure et aller plus loin</b>	<b>13</b>
<b>7</b>	<b>Lignes de métro</b>	<b>14</b>

## 1 Introduction

Ce rapport décrit le projet C++ de Kim Antunez et d'Alain Quartier-la-Tente (Ensaе, 2A) dont l'objectif est de permettre à l'utilisateur d'obtenir un itinéraire entre deux stations de métro selon deux critères : le plus court chemin ou le chemin avec le moins de correspondances. L'ensemble des données et des codes utilisés sont disponibles sous [https://github.com/AQLT/Metro\\_Cpp](https://github.com/AQLT/Metro_Cpp), la section 2 décrivant comment utiliser l'application. Les données utilisées sont les données du métro parisien fournies par la RATP (section 3), l'implémentation des classes est décrite dans la section 4, l'algorithme utilisé pour calculer les chemins est l'algorithme de Dijkstra (section 5). Enfin, une conclusion et quelques pistes d'amélioration sont décrites dans la partie 6.

Pour mieux se retrouver dans le réseau parisien, et surtout tester la validité de l'algorithme, le plan de l'ensemble des lignes a été rajouté dans la section 7.

## 2 Le programme

### 2.1 Prise en main

Afin de s'assurer que le programme tourne bien sur votre ordinateur, veuillez à effectuer les vérifications suivantes dans le fichier `main.cpp` :

1. Changer le répertoire de travail (`project_directory`) : il doit s'agir du lien vers le dossier contenant notamment le dossier `Data Projet`. Ainsi, si l'archive a été dézippée sous `D:/`, `project_directory` doit être égale à `D:/Metro_Cpp`.
2. Choisir si l'on veut mettre de la couleur dans l'interface (booléen `activerCouleur`).



**Attention** : la coloration de la console utilise la librairie `windows.h` et n'est donc compatible qu'avec les ordinateurs sous Windows. Si vous utilisez un autre système d'exploitation, ouvrez le fichier `IHM.h` et supprimer la ligne 6 contenant `#include <windows.h>`.

3. Lancer le programme et se laisser guider.

```
int main()
{
    // ATTENTION PENSER A CHANGER !
    // Mettre le lien vers le dossier qui contient le dossier Data projet
    string project_directory = "W:/Documents/Cplusplus/Projet/ProjetC"; // Kim
    //string project_directory = "W:/Bureau/Exercices/ExerciceC"; //Alain
    bool activerCouleur = true; //Pour activer la coloration syntaxique
    // ATTENTION : si on n'utilise pas Windows, supprimer la ligne 6 du fichier
    // IHM.h contenant "#include <windows.h>"

    IHM menu(activerCouleur); // on crée un objet menu
    Metro metro; // on crée un objet metro
    metro.importerDonnees(project_directory); // on importe les données
    Graphe graphe(project_directory); // on crée un objet graphe
    vector<string> identifiants_depart_arrivee = menu.choixDepartArrivee(metro);
    // On affiche le menu de choix de la station de depart et d'arrivee et on récupère les 2 identifiants de stations
    bool minChangement = menu.choixTypeItineraire(); // on affiche le menu de type d'itineraire (Chemin le plus rapide
    // ou minimum de changements) et on récupère la réponse
    Itineraire itineraire_sortie(graphe.dijkstras(identifiants_depart_arrivee[0],
    identifiants_depart_arrivee[1],
    minChangement),
    metro); // On calcule l'itineraire le plus court en fonction de ce choix grâce à l'algo
    menu.afficherItineraire(itineraire_sortie); // On affiche l'itineraire
```

Changer ces deux variables et éventuellement le fichier `IHM.h` si on n'est pas sous Windows.

### 2.2 Démonstration

Nous cherchons ici à calculer l'itinéraire le plus court entre la station de métro **Gabriel-Péri** (sur la **ligne 13**) et la station **Château de Vincennes** (sur la **ligne 1**). Le résultat (critère du plus court chemin et du moins de correspondances) est illustré en figure 1.

Pour cela, il suffit de se laisser guider par le menu en indiquant tout d'abord dans la console le numéro (parfois suivi de "B" pour les lignes bis) de la ligne de la station de départ, puis le code du menu qui correspond au nom de la station. Il faut ensuite refaire les mêmes actions pour le choix de la station d'arrivée.

```

-----
----- MENU -----
-----

Choisissez votre ligne de depart parmi les lignes suivantes :
1, 2, 3, 3B, 4, 5, 6, 7, 7B, 8, 9, 10, 11, 12, 13 et 14
Ligne de depart : 13

Choisissez votre arret de depart :

0 : Asnieres-Gennevilliers Les Courtilles
1 : Basilique de Saint-Denis
2 : Brochant
3 : Carrefour-Pleyel
4 : Champs-Elysees-Clemenceau
5 : Chatillon Montrouge
6 : Duroc
7 : Gabriel-Peri
8 : Gaite
9 : Garibaldi
10 : Guy-Moquet
11 : Invalides
12 : La Fourche
13 : Les Agnettes
14 : Liege
15 : Mairie de Clichy
16 : Mairie de Saint-Ouen
17 : Malakoff-Plateau de Vanves
18 : Malakoff-Rue Etienne Dolet
19 : Miromesnil
20 : Montparnasse-Bienvenue
21 : Pernet
22 : Place de Clichy
23 : Plaisance
24 : Porte de Clichy
25 : Porte de Saint-Ouen
26 : Porte de Vanves
27 : Saint-Denis - Porte de Paris
28 : Saint-Denis-Universite
29 : Saint-Francois-Xavier
30 : Saint-Lazare
31 : Varenne

Arret de depart : 7
(Gabriel-Peri)

Choisissez votre ligne d'arrivee parmi les lignes suivantes :
1, 2, 3, 3B, 4, 5, 6, 7, 7B, 8, 9, 10, 11, 12, 13 et 14
Ligne d'arrivee : 1

Choisissez votre arret d'arrivee :

0 : Argentine
1 : Bastille
2 : Berauld
3 : Champs-Elysees-Clemenceau
4 : Charles de Gaulle-Etoile
5 : Chateau de Vincennes
6 : Chatelet
7 : Concorde
8 : Esplanade de la Defense
9 : Franklin-Roosevelt
10 : Gare de Lyon
11 : George V
12 : Hotel de Ville
13 : La Defense (Grande Arche)
14 : Les Sablons (Jardin d'acclimatation)
15 : Louvre-Rivoli
16 : Nation
17 : Palais-Royal (Musee du Louvre)
18 : Pont de Neuilly
19 : Porte Maillot
20 : Porte de Vincennes
21 : Reuilly-Diderot
22 : Saint-Mande
23 : Saint-Paul (Le Marais)
24 : Tuileries

Arret d'arrivee : 5
(Chateau de Vincennes)

Voulez-vous l'itineraire :
0 : Le plus rapide ?
1 : Avec le moins de changements ?
0

```

Il est ensuite possible de choisir le **type d'itinéraire** souhaité. Nous choisissons ici le **chemin le plus rapide**.

La console nous indique donc le chemin le plus court.

```
-----
----- Votre itineraire -----
-----

A Gabriel-Peri prendre la ligne 13 direction CHATILLON - MONTROUGE jusqu'a l'arr
et Saint-Lazare (7 arrêts), puis prendre la ligne 14 direction OLYMPIADES jusqu'
a l'arrêt Gare de Lyon (4 arrêts), puis prendre la ligne 1 direction CHATEAU DE
VINCENNES jusqu'a l'arrêt Chateau de Vincennes (6 arrêts)

Temps de trajet minimum : 36 min

Voulez-vous :
0 : Quitter l'application ?
1 : Chercher un nouvel itineraire ?
1
```

“ À Gabriel-Péri, prendre la ligne 13 direction Châtillon-Montrouge jusqu'à l'arrêt Saint-Lazare (7 arrêts), puis prendre la ligne 14 direction Olympiades jusqu'à l'arrêt Gare de Lyon (4 arrêts), puis prendre la ligne 1 direction Château de Vincennes jusqu'à l'arrêt Château de Vincennes (15 arrêts).

”

Si l'on change le **type d'itinéraire** en choisissant le **chemin avec le minimum de correspondances**, on obtient l'itinéraire suivant.

```
Voulez-vous l'itineraire :
0 : Le plus rapide ?
1 : Avec le moins de changements ?
1

-----
----- Votre itineraire -----
-----

A Gabriel-Peri prendre la ligne 13 direction CHATILLON - MONTROUGE jusqu'a l'arr
et Champs-Elysees-Clémenceau (9 arrêts), puis prendre la ligne 1 direction CHATE
AU DE VINCENNES jusqu'a l'arrêt Chateau de Vincennes (15 arrêts)

Temps de trajet minimum : 40 min

Voulez-vous :
0 : Quitter l'application ?
1 : Chercher un nouvel itineraire ?
=
```

“ À Gabriel-Péri, prendre la ligne 13 direction Châtillon-Montrouge jusqu'à l'arrêt Champs-Elysées-Clémenceau (9 arrêts), puis prendre la ligne 1 direction Château de Vincennes jusqu'à l'arrêt Château de Vincennes (15 arrêts).

”

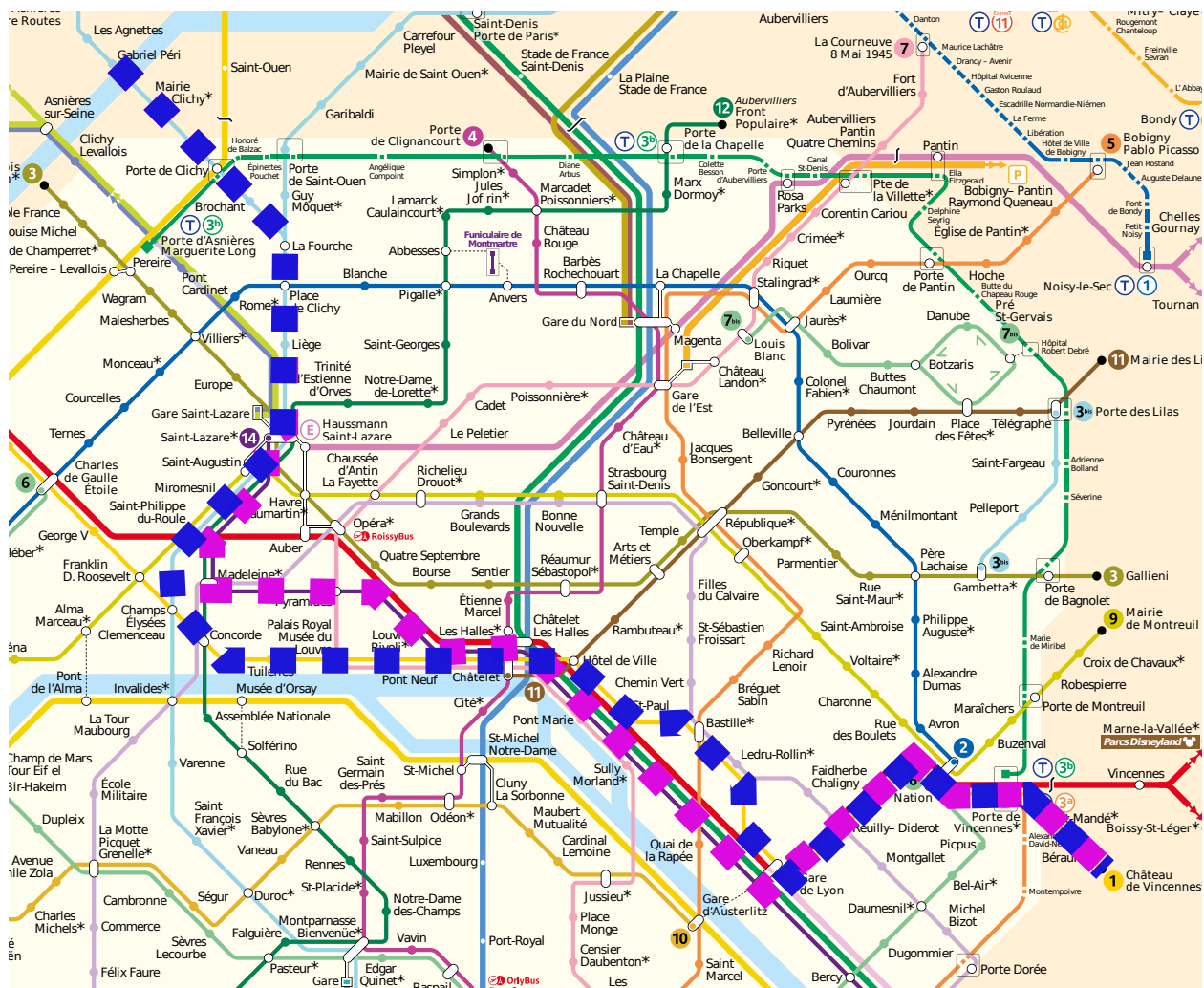


FIGURE 1: Itinéraires de Gabriel-Péri à Château de Vincennes

Note de lecture : Le chemin le plus court est représenté en rose et celui avec le minimum de changements en bleu.

## 3 Description des données

### 3.1 Description des données à disposition

Toutes les données utilisées dans ce projet sont issues de la RATP, plus précisément de la base `offre-transport-de-la-ratp-format-gtfs` (<https://dataratp.opendatasoft.com/explore/dataset/offre-transport-de-la-ratp-format-gtfs/information/>). Ces données sont au format *General Transit Feed Specification* (GTFS) qui est un format standardisé pour diffuser des données relatives aux réseaux de transport en commun (horaires, informations géographiques, etc.).

Deux archives de données de la RATP sont mises à disposition :

- Une archive avec des fichiers GTFS répartis par lignes ;
- Une archive avec des fichiers GTFS pour l'ensemble du réseau (métro, bus, tram et RER).

C'est la première archive que nous avons utilisée. En effet, nous restreignons notre étude aux lignes de métro uniquement. Nous avons stocké les données utilisées ici : [https://github.com/AQLT/Metro\\_Cpp/tree/master/Data](https://github.com/AQLT/Metro_Cpp/tree/master/Data).

Chaque ligne de métro est associée à un dossier qui contient des données stockées dans différents fichiers :

1. `routes.txt` : définit les itinéraires des transports en commun → données **utilisées** avec le fichier `trips.txt` pour identifier l'ordre de passage à chaque arrêt pour les lignes aller et retour.
2. `stops.txt` : définit l'ensemble des arrêts où les usagers peuvent monter ou descendre, avec le nom de l'arrêt, l'adresse et les coordonnées GPS → données **utilisées** dans ce projet pour définir l'ensemble des arrêts.
3. `stop_times.txt` : définit, pour chaque trajet et pour chaque arrêt, les heures d'arrivée et de départ du métro → données **utilisées** avec le fichier `trip.txt` pour connaître le temps de trajet entre deux stations d'une même ligne.
4. `transfers.txt` : définit les règles de liaison aux pôles de correspondance entre des itinéraires → données **utilisées** dans ce projet pour connaître les correspondances et les temps de correspondance entre les lignes.
5. `trips.txt` : définit l'ensemble des trajets pour chaque ligne (i.e. : tous les trajets prévus dans la journée) → données **utilisées** avec le fichier `routes.txt` et `stop_times.txt` pour identifier pour chaque ligne l'ordre de passage et le temps de trajet entre chaque arrêt <sup>1</sup>.
6. `calendar_dates.txt` : définit les exceptions pour les services définis dans le fichier `calendar.txt` → données **non utilisées** dans ce projet.
7. `calendar.txt` : définit les dates auxquelles le service est disponible pour des itinéraires spécifiques selon un calendrier hebdomadaire. Ce fichier spécifie les dates de début et de fin du service, ainsi que les jours de la semaine où le service est disponible → données **non utilisées** dans ce projet.
8. `agency.txt` : définit une ou plusieurs agences de transports publics dont les services sont représentés dans l'ensemble de données → données **non utilisées** dans ce projet.

Plus d'informations sur les données GTFS sont disponibles sur le site de Google : <https://developers.google.com/transit/gtfs/reference/>.

### 3.2 Difficultés et solutions adoptées

Chaque arrêt est défini par un identifiant unique. Cet identifiant est différent pour chaque ligne et pour chaque route (aller et retour). Certains itinéraires étaient donc impossibles à calculer avec l'utilisation de ces seules données brutes. Par exemple, si l'on est à l'arrêt de métro Gaîté sur la ligne 13 direction Châtillon-Montrouge on ne peut pas rejoindre l'arrêt Montparnasse car la ligne est orientée dans la "mauvaise direction".

---

1. Dans ce projet nous ne prenons pas en compte l'heure à laquelle la recherche d'itinéraire a été faite : pour chaque "route" un seul "trip" a donc été utilisé.

→ **Solution adoptée** : ajouter un temps de transfert égal à 0 permettant de passer d'un arrêt d'une ligne aller à ce même arrêt (même nom) de la ligne retour, et inversement. Ainsi, le temps de transfert est nul pour passer de l'arrêt de métro Gaîté sur la ligne 13 direction Châtillon-Montrouge à l'arrêt de métro Gaîté sur la ligne 13 direction Saint-Denis/Les Courtilles.


Toutefois, même avec cette correction des données, des limites concernant les temps de transfert subsistent. Par exemple, pour réaliser certains itinéraires, il est nécessaire de changer de direction tout en restant sur la même ligne (par exemple sur la ligne 13 passer de Guy Môquet à Brochant il faut aller jusqu'à l'arrêt La Fourche et changer de direction).

Puisque nous avons fait le choix de ne pas tenir compte du temps d'attente moyen d'un métro, **le temps de trajet prévu pour chaque itinéraire est sous-estimé.**

Cette façon de numérotiser les arrêts implique que certains arrêts ne sont associés qu'à une ligne (aller ou retour) alors que d'autres sont associés à deux lignes (par exemple sur la ligne 13 qui contient 2 lignes aller ou 2 lignes retour).

Plusieurs incohérences ont également été corrigées dans les données :

- Le fichier `routes.txt` ne permettait pas toujours de bien identifier les lignes aller et retour. En effet, pour certaines "routes" (par exemple : ligne 13 de Châtillon-Montrouge à Saint-Denis/Les Courtilles), certains départs étaient identifiés comme partant du terminus opposé. Cela devrait normalement être impossible puisque la "route" permet d'identifier la direction. Ce problème affecte les lignes 1, 4, 7, 7B et 13 : un travail manuel sur les bases de données a donc été réalisé pour identifier correctement les lignes aller et retour.
- Certaines "routes" de la base de données ne correspondent pas à la réalité du réseau du métro parisien. C'est le cas d'une des routes de la ligne 10 "BOULOGNE - PONT DE SAINT CLOUD <-> GARE D'AUSTERLITZ) - Aller" qui partirait de l'arrêt Porte d'Auteuil pour ensuite aller à l'arrêt Michel-Ange Molitor et continuer direction Gare D'Austerlitz (alors que depuis Porte d'Auteuil la seule direction possible est Boulogne). Cette "route" n'a alors pas été considérée dans notre algorithme.

Pour simplifier le chargement des données en C++, nous avons pré-traité les données via le logiciel statistique  :

- Pour chaque ligne de métro nous avons créé un dossier avec :
  - Le fichier `stops.txt` dans lequel nous avons enlevé les accents (permet de créer l'ensemble des arrêts)<sup>2</sup> ;
  - Un fichier par "route" contenant l'ensemble des arrêts de manière ordonnée ainsi que le nom de la route (Ex : CHATEAU DE VINCENNES <-> LA DEFENSE - Aller) , ce qui permet d'identifier les arrêts traversés par une ligne ainsi que l'ordre de passage.
- Deux matrices carrées ayant autant de colonnes et de lignes que d'identifiants d'arrêts :
  - `voisins_type.txt` : la coordonnée (i,j) vaut -1 si les deux arrêts ne sont pas directement connectés, 0 si les deux arrêts sont voisins et sur une même ligne et 1 si les deux arrêts sont connectés mais sur deux lignes différentes (par exemple entre la ligne 4 et la ligne 13 à l'arrêt Montparnasse).
  - `voisins.txt` : la coordonnée (i,j) correspond au temps nécessaire pour aller directement l'arrêt i à l'arrêt j. Avec une valeur égale à -1 s'il n'y a pas de correspondance directe possible et à 0 si les deux arrêts en fait "les mêmes" (i.e. : ce sont des arrêts qui ont le même nom et le même numéro de ligne, cf. plus haut).

Les fichiers utilisés dans l'implémentation C++ sont disponibles sous [https://github.com/AQLT/Metro\\_Cpp/tree/master/Data%20projet](https://github.com/AQLT/Metro_Cpp/tree/master/Data%20projet).

---

2. le logiciel Code::Blocks utilisé sur les postes Windows de l'Ensaë génère en effet des problèmes d'encodage, ce qui ne permet pas par exemple d'afficher correctement les caractères accentués.

## 4 Description des classes

Nous avons cherché à rendre l’implémentation de ce projet C++ la plus modulable possible en séparant notamment les classes relatives à l’algorithme du plus court chemin de celles relatives à l’architecture du réseau de métro parisien. Nous avons créé huit classes pouvant être rassemblées en quatre groupes :

1. Les classes liées à la représentation des données du métro parisien. Il s’agit des classes **Arret**, **Ligne** et **Metro**.
2. Les classes liées à l’algorithme du plus court chemin. Il s’agit des classes **Node**, **Edge** et **Graphe**.
3. Une classe permettant de faire le lien entre les données et l’algorithme. Il s’agit de la classe **Itineraire**.
4. Une classe générant l’Interface Homme-Machine. Il s’agit de la classe **IHM**.

La figure 2 décrit les relations entre toutes ces classes. Elle reprend un certain nombre de conventions d’un diagramme de classe en UML (objets de type public préfixés d’un “+” et ceux de type privé d’un “-”) sans pour autant respecter toutes ses normes. Par exemple, les constructeurs ne figurent pas sur ce schéma<sup>3</sup>.

Des informations supplémentaires sur les méthodes mobilisées dans chaque classe sont disponibles dans le code du projet.

### 4.1 Classes liées au réseau de métro parisien

À partir des données décrites dans la section 3, nous avons créé quatre classes :

1. La classe **Arret** est la classe qui représente un arrêt de métro tel que défini dans la base de données de la RATP : il dépend donc du numéro de la ligne ainsi que de sa direction. Il y a donc par exemple deux objets **Arret** associés à l’arrêt “Gaîté” (un qui correspond à l’arrêt Gaîté sur la ligne 13 direction Châtillon-Montrouge et un qui correspond à l’arrêt Gaîté sur la ligne 13 direction Saint-Denis/Les Courtilles) et quatre objets **Arret** associés à l’arrêt Denfert-Rochereau (deux sur la ligne 4 et deux sur la ligne 5). Chaque arrêt peut être associé à un ou plusieurs objets **Ligne**.
2. La classe **Ligne** représente un ensemble ordonné d’objets **Arret** associés à une même ligne (même numéro de ligne mais également même direction). Ainsi, pour la ligne 4 il y a deux objets **Ligne** qui correspondent au chemin de la ligne 4 direction Mairie de Montrouge et au chemin de la ligne 4 direction Porte de Clignancourt. De la même façon, pour la ligne 13, il y a quatre objets **Ligne**.
3. La classe **Metro** est la classe qui synthétise le réseau de métro à Paris : elle contient l’ensemble des objets **Arret** et l’ensemble des objets **Ligne**.

### 4.2 Classes liées à l’algorithme du plus court chemin

Trois classes ont été créées pour implémenter l’algorithme de Dijkstra :

1. La classe **Node** est la classe qui représente un sommet (au sens de la théorie des graphes). Chaque sommet est associé à un identifiant (qui correspond à un identifiant d’un **Arret**) et possède plusieurs paramètres qui seront actualisés pendant l’algorithme de Dijkstra (voir section 5).
2. La classe **Edge** est la classe qui représente une arête. C’est-à-dire qu’un **Edge** représente un lien orienté d’un **Node** 1 vers un **Node** 2 avec la distance entre ces deux objets qui vaut :
  - soit le temps de correspondance/temps de trajet entre deux arrêts si l’on souhaite calculer l’itinéraire le plus court
  - soit la valeur de l’indicatrice d’être sur la même ligne si l’on souhaite calculer l’itinéraire avec le moins de correspondances.
3. La classe **Graphe** est la classe qui synthétise le réseau du côté algorithmique : elle contient l’ensemble des **Node** et des **Edge** et permet de calculer le plus court chemin.

---

3. à l’exception d’un des constructeurs de la classe **Itineraire** qui met en évidence le lien d’**Itineraire** avec non seulement le réseau de métro (**Metro**, **Arret**) mais également l’algorithme (**Node**)



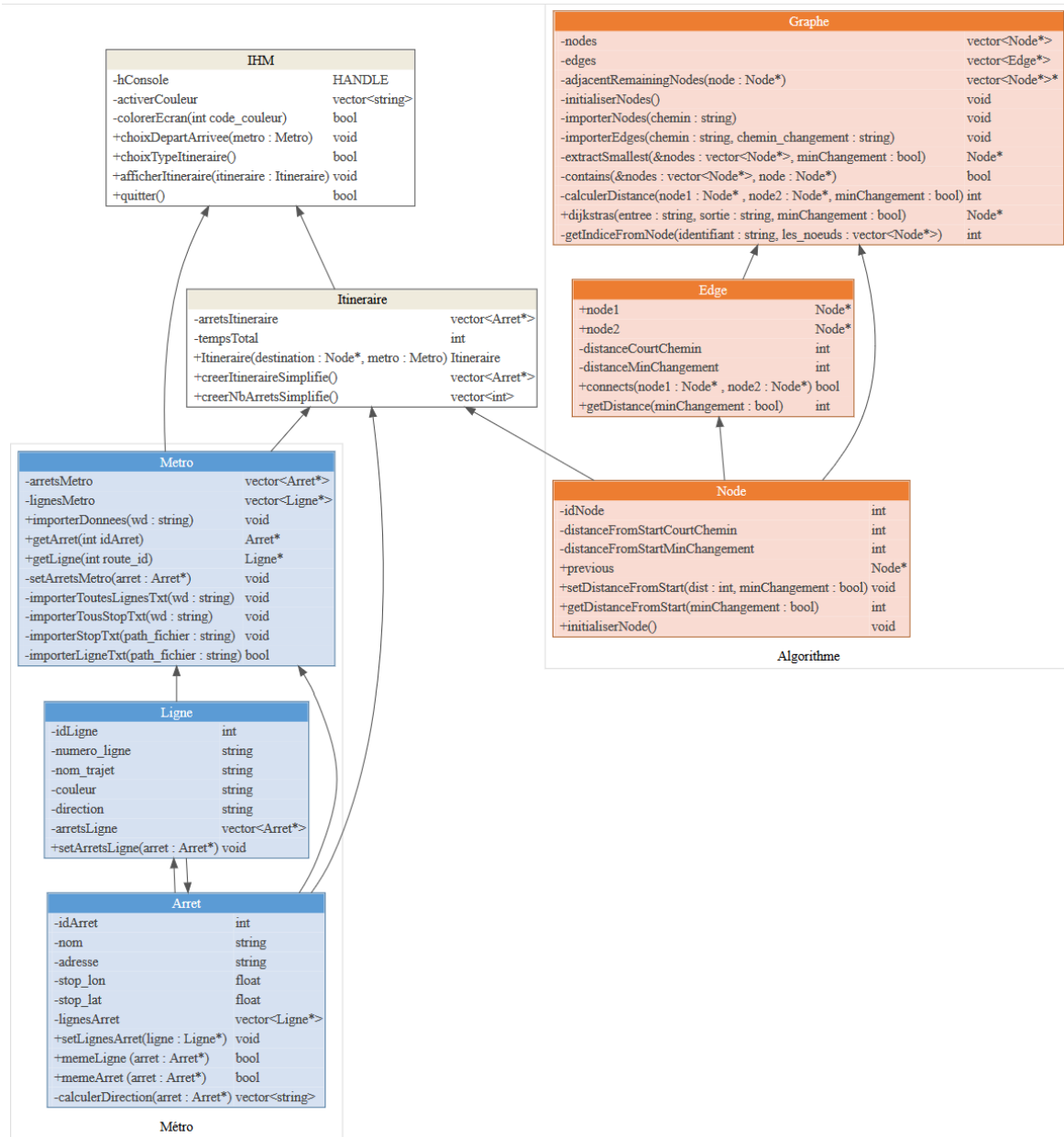


FIGURE 2: Liens entre les classes du projet

Note de lecture : Les objets de type public sont préfixés d'un "+" et ceux de type privé d'un "-". Les constructeurs ne figurent pas sur ce schéma, à l'exception d'un des constructeurs d'*Itineraire*.

### 4.3 Classe faisant le lien entre l'algorithme et les données

La classe *Itineraire* permet de faire le lien entre les données et le résultat de l'algorithme du plus court chemin. Son constructeur utilise le *Node* d'arrivée et le *Metro* pour :

- retracer l'ensemble des *Node* parcourus dans le plus court chemin et en déduire l'ensemble des *Arret* parcourus durant l'itinéraire ;
- calculer le temps nécessaire pour effectuer le trajet.

Elle fournit également quelques fonctions pour faciliter l'affichage du résultat par l'*IHM*.

## 4.4 Classe générant l'Interface Homme-Machine

Une classe IHM a été créée pour gérer l'interface homme-machine. Elle contient quatre grandes fonctionnalités qui correspondent à un découpage de l'affichage du menu en quatre parties :

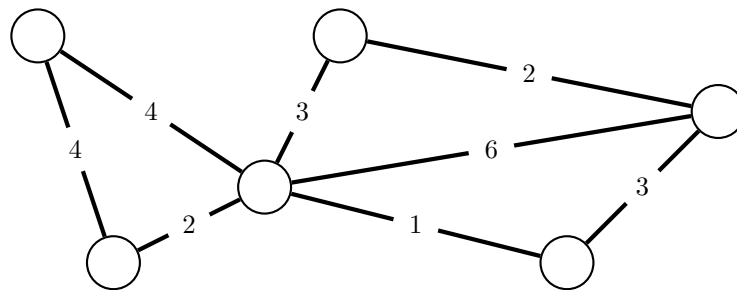
- `choixDepartArrivee()` qui, à partir d'un objet `Metro`, permet à l'utilisateur de choisir sa station de métro de départ puis d'arrivée ;
- `afficherItineraire()` qui, à partir d'un objet `Itineraire`, récupère les principales informations à afficher à l'utilisateur concernant son itinéraire (station de départ, d'arrivée, correspondances, temps de trajet...);
- `choixTypeItineraire()` qui permet à l'utilisateur de choisir un type d'itinéraire (plus court chemin ou le moins de correspondances)
- `quitter()` qui permet à l'utilisateur de quitter l'application.

## 5 L'algorithme de Dijkstra

L'algorithme de Dijkstra permet de trouver le chemin le plus court entre deux sommets d'un graphe.

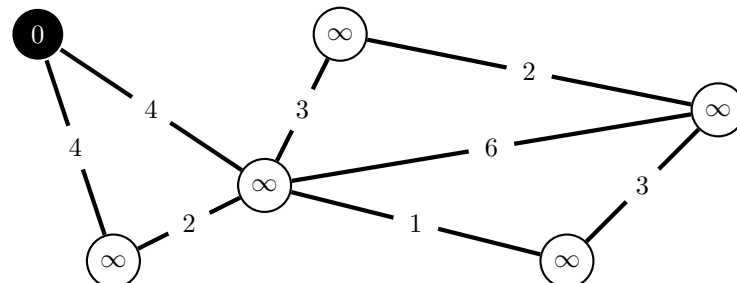
Par simplification, l'algorithme est ci-dessous présenté grâce à un exemple fictif, qui fait également référence aux **fonctions**, **paramètres** et **objets** utilisés dans notre code.

1. En entrée, nous partons d'un graphe (**Graphe**) composé de sommets (**Node**) reliés par des arêtes (**Edge**) auxquelles on associe une distance.



*Remarque :* Dans l'exemple choisi ici, le graphe n'est pas orienté : si le sommet A est relié au sommet B, cela implique que B est aussi relié à A et la distance de A vers B est égale à celle de B vers A. Dans les données de la RATP que nous utilisons, le graphe est, lui, orienté. Toutefois, l'algorithme fonctionne de la même façon dans ce cadre.

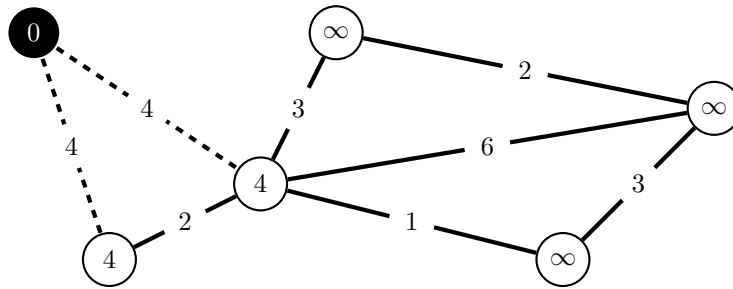
2. Choisir un sommet de départ. Lui attribuer un attribut de distance (`Node.distanceFromStart`) égal à 0. Attribuer une valeur infinie à `distanceFromStart` pour tous les autres sommets.



*On s'intéresse tout d'abord au sommet le plus à gauche coloré en noir.*

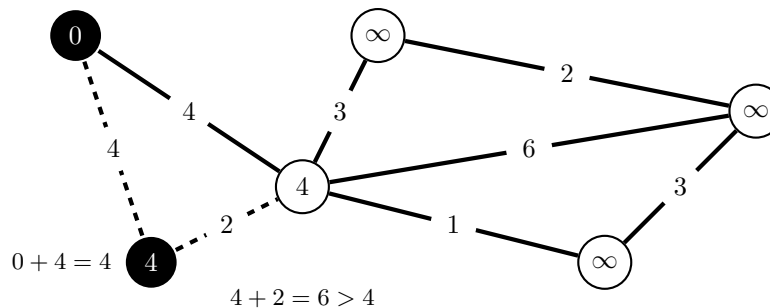
3. Pour tous les sommets adjacents à ce sommet de départ, actualiser le `distanceFromStart` du sommet avec la valeur égale à la distance entre le sommet de départ et ce sommet. Pour chaque nouveau

sommet parcouru, on enregistre le sommet précédemment parcouru ([Node.previous](#)) afin de pouvoir, in fine, retracer l'itinéraire parcouru.



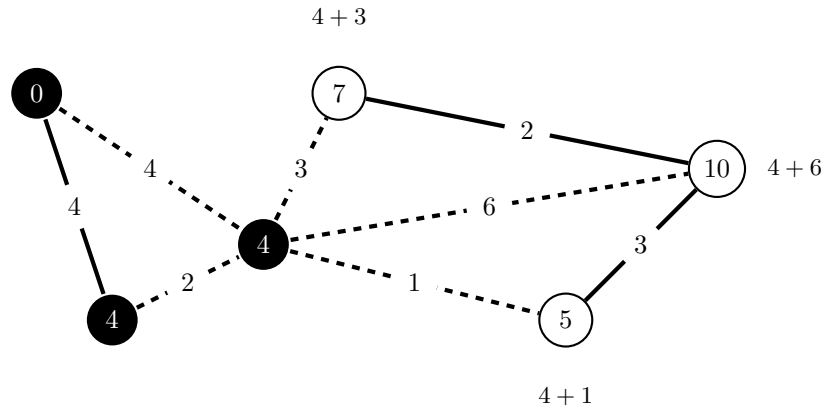
*Ici on actualise donc la valeur du distanceFromStart des deux sommets voisins du sommet de départ à la valeur de 4. Le Node.previous de ces deux voisins est égal au sommet de départ.*

4. Choisir un « sommet voisin » (nous allons voir en étape 6 comment précisément le choisir). Sommer ensuite la valeur de la distanceFromStart du « sommet précédent » à la distance entre ce « sommet précédent » et le « sommet voisin » (cette somme vaut `dist = calculerDistance()`). Si l'ancienne distanceFromStart associée au « sommet voisin » est supérieure à « dist » alors on la modifie à « dist », sinon on la laisse inchangée. Puis, on passe au sommet suivant.



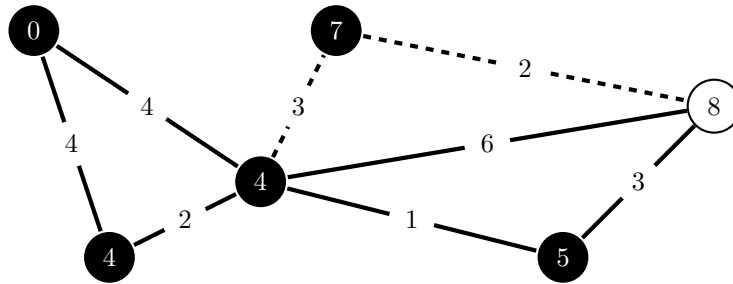
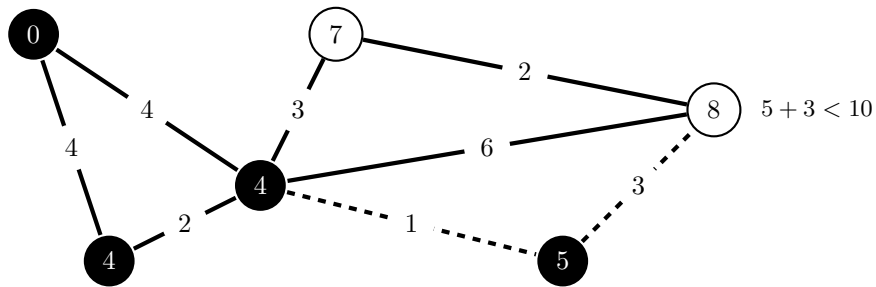
Ici, on s'intéresse par exemple dans un deuxième temps au sommet en bas à gauche puis dans un troisième temps à son voisin de droite. On laisse la valeur de 4 à ce voisin de droite car  $4+2=6 > 4$ .

- Continuer à parcourir le graphe. À chaque itération, identifier tout d'abord les voisins restant à parcourir (`adjacentRemainingNodes()`) et choisir le sommet non visité pour lequel la `distanceFromStart` est la plus petite (`extractSmallest()`).



En troisième étape, on calcule les *distanceFromStart* de tous les voisins du sommet central non parcourus. Puis en quatrième étape, nous étudions le sommet (5) avant les sommets (7) et (10) ( $5 < 7 < 10$ ).

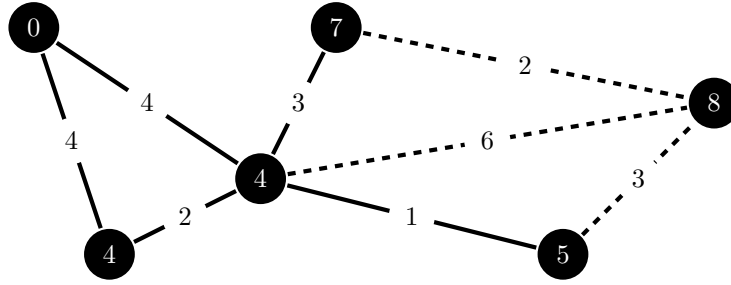
6. Continuer à parcourir le graphe.



Le *distanceFromStart* du sommet le plus à droite est alors mis à jour deux fois.

7. Enregistrer les chemins parcourus à chaque étape et répéter jusqu'à ce que toutes les arêtes soient visitées<sup>4</sup>. Il est alors possible en choisissant un sommet d'arrivée de retracer le chemin le plus court à parcourir et les distances parcourues entre chaque sommet.

4. On pourrait spécialiser l'algorithme en arrêtant de parcourir le graphe une fois le sommet d'arrivée atteint. Toutefois, le temps de calcul étant relativement faible, le gain serait alors limité.



*L'algorithme se termine : tous les sommets sont parcourus.*

L'implémentation choisie dans le cadre de ce projet reprend ces différentes étapes. Il ne s'agit pas de la méthode la plus optimale en termes de temps de calcul mais, en revanche, de la méthode la plus lisible (classes et fonctions plus facilement compréhensibles). Par ailleurs, du fait de la petite taille du graphe utilisé dans le projet, le gain qui découlerait d'une optimisation du temps de calcul serait négligeable.

## 6 Pour conclure et aller plus loin

Nous avons tous deux beaucoup apprécié travailler sur ce projet. Nous avons découvert la programmation objet avec Java il y a quelques années mais avons peu eu l'occasion de mobiliser ces connaissances depuis. Ce projet a donc été l'occasion de consolider ces acquis. Par ailleurs, l'objectif du projet nous a semblé très complet puisqu'il nous a permis de travailler à la fois sur un **algorithme**, une **Interface Homme-Machine** et un **diagramme de classes relativement riche**.

Toute bonne chose ayant une fin, voici donc quelques pistes qui pourraient permettre d'aller plus loin :

- Le format matriciel utilisé pour stocker les temps de correspondance et de trajet n'est pas optimal. En effet, ce format conduit à lire des informations inutiles : celles qui concernent les stations de métro non reliées, qui sont nombreuses (matrice creuse). De plus, pour charger les deux types de distance, deux matrices sont alors parcourues. Le temps de chargement des données pourrait être amélioré de façon conséquente en créant un fichier dont chaque ligne contiendrait deux identifiants d'arrêts et les deux distances qui les séparent (le plus court chemin et le moins de correspondance).
- Nous n'avons pas pris en compte les horaires des métros dans ce projet. C'est par exemple ce qui explique que nous n'avons pas intégré les lignes de RER. Une façon de gérer l'heure (et la date) de passage des métros serait de rajouter dans la classe `Edge` un vecteur contenant l'ensemble des horaires de passage. Pendant la mise à jour des distances des `Node` de l'algorithme, il faut alors :
  1. mettre à jour la valeur de `Node.distanceFromStart` comme étant égale à la somme entre le temps de correspondance et la différence entre l'heure du prochain passage et l'heure actuel ;
  2. incrémenter l'heure actuel de cette somme.
- Lorsque deux itinéraires ont un temps de trajet "proche" (par exemple quand le temps de trajet est inférieur à 2 minutes), on pourrait proposer les deux possibilités à l'utilisateur.

D'autres informations pourraient être utilisées pour proposer d'autres fonctionnalités. Par exemples les coordonnées GPS (disponibles dans la classe `Arret`) pourraient être utilisées pour :

- Calculer l'itinéraire en prenant en compte le temps de marche nécessaire pour aller à la station de métro la plus proche.
- Prendre en compte le réseau de bus, de RER et de tramways.
- Ajouter d'autres types d'itinéraires : accessibles aux personnes en situation de handicap, ou encore passant par des toilettes publiques de la RATP (<https://datarotp.opendatasoft.com/explore/dataset/sanitaires-reseau-ratp/>) !

## 7 Lignes de métro

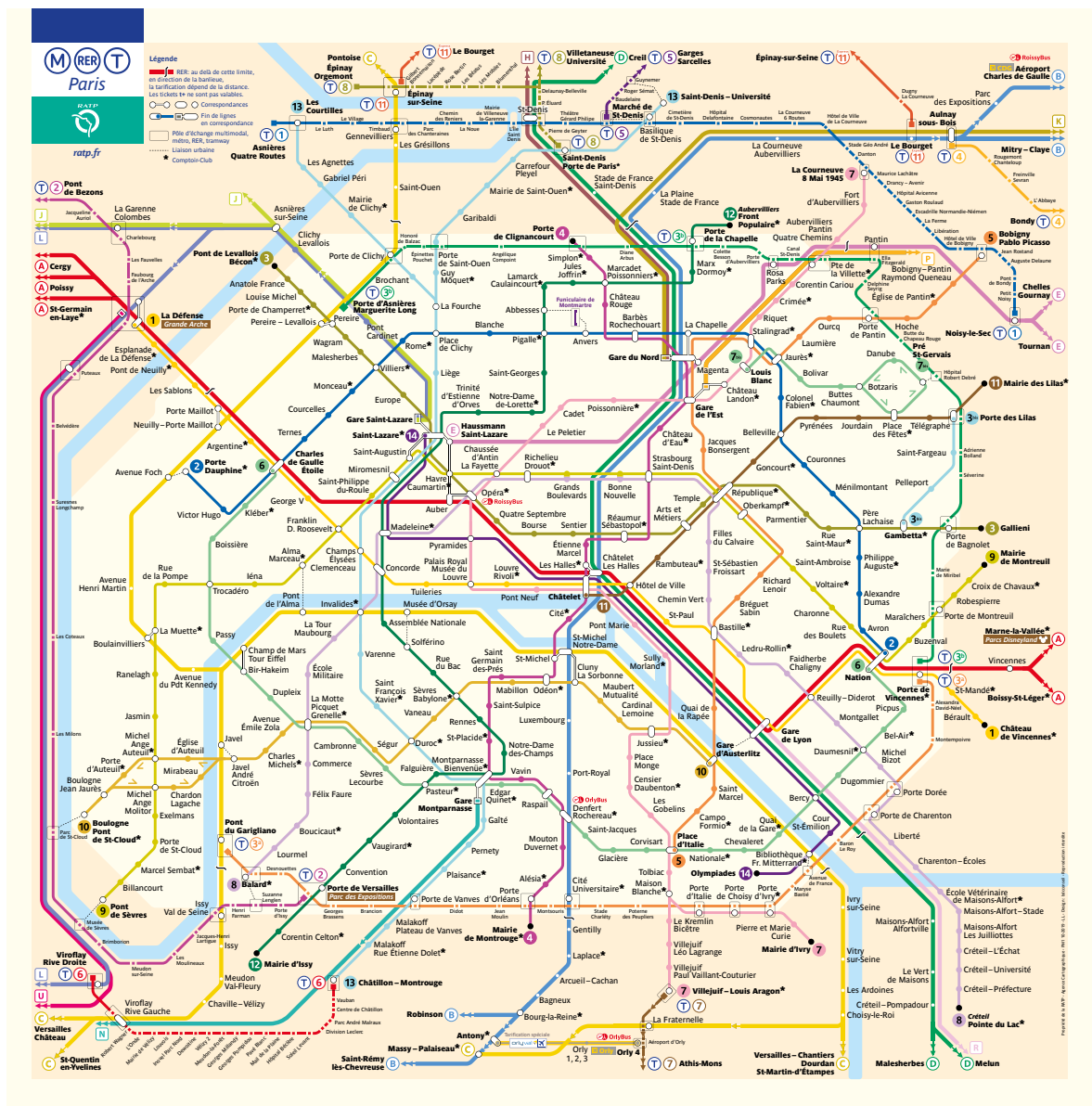


FIGURE 3: Plan général du réseau de transport parisien (hors bus)

